Assignment:

Data Structure:

In [ ]:

1. String Slicing in Python

- String slicing in Python allows extracting a portion of a string by specifying a range of indices. Strings are indexed starting from 0, and slicing uses the syntax string[start:stop:step].
- start: The index where the slice begins (inclusive).
- stop: The index where the slice ends (exclusive).
- step: Controls how the index progresses (optional).

In [1]:
```
text = "md akif nawab"
print(text[0:6])
```

```
md aki
```

In [2]:
```
print(text[7:18])

print(text[::-1])
```

```
 nawab
bawan fika dm
```

In [ ]:

2. Key Features of Lists in Python

Lists are one of the most versatile data structures in Python. Some key features include:

- Mutable: You can change, add, or remove elements after the list has been created.
- Ordered: The elements in a list are stored in a specific sequence.
- Dynamic Size: Lists can grow or shrink as elements are added or removed.
- Heterogeneous: Lists can hold elements of different data types.
- Accessing, Modifying, and Deleting Elements in a List
  - Access: Lists can be accessed using indices, starting from 0. - Modifying: You can update the value of a list element by assigning a new value to its index.
  - Deleting: Use del, remove(), or pop() to remove elements

In [3]:
```
my_list = [1, 20, 100, "akif", True, 2+4j]

# Access
print(my_list[1])
```

```
20
```

In [4]:
```
# Modify
my_list[1] = 200
print(my_list)
```

```
[1, 200, 100, 'akif', True, (2+4j)]
```

In [5]:
```
# Delete
del my_list[2]
print(my_list)
```

```
[1, 200, 'akif', True, (2+4j)]
```

In [ ]:

3.Working with Lists in Python: Accessing, Modifying, and Deleting Elements¶

- Lists in Python are a versatile and commonly used data structure. They allow you to store and manipulate a collection of items. This

section will cover how to access, modify, and delete elements in a list with examples.

- Accessing Elements

  -To access elements in a list, you use indexing. Python lists are zero-indexed, meaning the index starts at 0.

- Modifying Elements
  - You can modify elements in a list by assigning a new value to a specific index.
- Deleting Elements
  - You can delete elements from a list using several methods: del, remove(), and pop().
- Using del:

The del statement removes an element at a specified index.

In [23]:
```
# Defining a list
my_list = [10, 20, 30, 40, 50]


print(my_list[0])
print(my_list[2])
print(my_list[-1])
```

```
10
30
50
```

In [24]:
```
my_list = [10, 20, 30, 40, 50]

# Modifying elements by index
my_list[1] = 25
print(my_list)
```

```
[10, 25, 30, 40, 50]
```

In [25]:
```
# Defining a list
my_list = [10, 20, 30, 40, 50]
del my_list[3]
print(my_list)
```

```
[10, 20, 30, 50]
```

In [26]:
```
my_list.remove(30)
print(my_list)
```

```
[10, 20, 50]
```

In [ ]:
```
```

4. Comparing Tuples and Lists

- Tuples:
  - Immutable: Once created, their elements cannot be modified.
  - Ordered: Elements are stored in a specific order.
  - Fixed Size: You cannot add or remove elements from a tuple after it is created.
  - Performance: Tuples are faster than lists due to immutability.
- Lists:
  - Mutable: Elements can be changed, added, or removed.
  - Ordered: The order of elements is maintained.
  - Dynamic Size: Lists can grow and shrink as needed.

In [6]:
```
my_tuple = (1, 2, 3)
my_list = [1, 2, 3]

# Attempt to modify

my_list[0] = 10
```

In [7]:
```
my_list
```

Out[7]:
```
[10, 2, 3]
```

In [8]:
```
my_tuple[0] = 100
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[8], line 1
----> 1 my_tuple[0] = 100

TypeError: 'tuple' object does not support item assignment
```

In [ ]:

5. Key Features of Sets in Python

- Sets are unordered collections of unique elements.
  - Unordered: Elements do not have a fixed position.
  - Unique Elements: Duplicate values are automatically discarded.
  - Mutable: Elements can be added or removed.
- Use Cases for Sets:
  - Sets: Useful when you need to ensure that all elements are unique, such as eliminating duplicates from a list.

In [9]:
```
my_set = {1, 2, 3, 4, 4}
print(my_set)

# Adding elements
my_set.add(5)
print(my_set)
```

```
{1, 2, 3, 4}
{1, 2, 3, 4, 5}
```

In [10]:
```
# Removing elements
my_set.remove(2)
print(my_set)
```

```
{1, 3, 4, 5}
```

In [16]:
```
my_set.pop()
```

Out[16]:
```
1
```

In [17]:
```
my_set
```

Out[17]:
```
{3, 4, 5}
```

In [ ]:

6.Use Cases of Tuples and Sets in Python Programming

- Tuples:
  - Fixed Data: Ideal for storing collections of items that should not change. For example, coordinates or fixed records.
  - Function Return Values: Useful for returning multiple values from a function.
  - Data Integrity: Ensures that the data remains unchanged, useful for constant data.
  - Dictionary Keys: Can be used as keys in dictionaries if all elements are immutable.
- Sets:
  - Removing Duplicates: Automatically filters out duplicate items from a collection.
  - Membership Testing: Efficiently checks for the presence of an item.
  - Set Operations: Supports operations like union, intersection, and difference.
  - Efficient Data Handling: Ideal for tasks requiring fast lookups and unique data.

In [ ]:

7. Adding, Modifying, and Deleting Items in a Dictionary

- Dictionaries are collections of key-value pairs. They are unordered and mutable, and their keys must be unique and immutable.

- Adding Items: Assign a value to a new key.
- Modifying Items: Update the value associated with an existing key.
- Deleting Items: Use del or pop() to remove key-value pairs.

In [18]:
```
my_dict = {"name": "akif", "age": 18}

# Adding an item
my_dict["city"] = "Hyderabad"
print(my_dict)
```

```
{'name': 'akif', 'age': 18, 'city': 'Hyderabad'}
```

In [19]:
```
# Modifying an item
my_dict["age"] = 26
print(my_dict)
```

```
{'name': 'akif', 'age': 26, 'city': 'Hyderabad'}
```

In [20]:
```
# Deleting an item
del my_dict["city"]
print(my_dict)
```

```
{'name': 'akif', 'age': 26}
```

In [ ]:
```

```

8. Importance of Dictionary Keys Being Immutable

- Dictionary keys must be immutable to maintain a consistent hash value, which is used to determine their placement in memory. Mutable objects like lists cannot be used as keys because their contents can change, affecting their hash value.

In [22]:
```
# Valid dictionary keys
my_dict = {1: "apple", "name": "Alice"}
```

In [ ]:
```

```

In [ ]:
```

```